Genome Homology Visualization by Short Similar Substring Enumeration (Acceleration and Visualization of Computation for Enumeration Problems)

AUTHOR(S):

Uno, Takeaki

CITATION:

Uno, Takeaki. Genome Homology Visualization by Short Similar Substring Enumeration (Acceleration and Visualization of Computation for Enumeration Problems). 数理解析研究所講究録 2009, 1644: 35-43

ISSUE DATE:

2009-04

URL:

http://hdl.handle.net/2433/140652

数理解析研究所講究録
第1644巻 2009年 35-43

Genome Homology Visualization by Short Similar Substring Enumeration

Takeaki Uno

uno@nii.jp, National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

Abstract. Finding similar substrings/substructures is a central task in analyzing huge amounts of genome data. In the sense of complexity theory, the existence of polynomial time algorithms for such problems is usually trivial since the number of substrings is bounded by the square of their lengths. However, straightforward algorithms do not work for practical huge databases because of their computation time of high degree order. This paper addresses the problems of finding pairs of strings with small Hamming distances from huge databases composed of short strings of a fixed length. Using this, we compare two genome sequences by solving this problem for all the fixed-length substrings taken from the sequences. We focus on the practical efficiency of algorithms, and propose an algorithm running in almost linear time of the database size. When there are so many similar pairs so that the visualization is impossible, we propose to use a filtering algorithm to remove the pairs which are not parts of similar long sequences. Computational experiments for genome sequences show the efficiency of the method. An implementation is available at the author's homepage[1]

1 Introduction

In this paper, we consider the problem of enumerating all pairs of similar strings in a set $\mathcal{S}$ of strings of the same length l. We can approach to general substring comparison problems through this problem since such non-short similar strings must include several such short similar substrings. As a similarity measure, we use Hamming distance. Thus the definition of the problem is as follows.

Short Hamming Distance String Pair Enumeration Problem
Input: a set $\mathcal{S}$ of strings of the same length l, a distance threshold d
Output: all pairs of strings S_1 and S_2 such that the Hamming distance between S_1 and S_2 is at most d.

We here call a pair of strings with Hamming distance at most d *similar string pair*. We consider the case in which the length l is small, and propose a practically efficient algorithm. The idea of the algorithm is to classify the strings in several ways so that two strings of similar string pair are in the same group for at least one classification. Only strings in the same group have to be compared, which reduces the cost of the comparison. Each string is partitioned into k blocks, then strings in a similar string pair must share at least $k-d$ same blocks. Thus, they are in the same group at least one classification based on combinations of $k-d$ of these blocks. We call this method "multi-sort algorithm". If the $k-d$ blocks sufficiently many letters, the members of each group is expected to be few, thus all pairs comparison takes quite short time, and practical computation time will be closed to linear time. As we show later, the result of the computational experiments for genome strings show that the algorithm is practically efficient. This algorithm can be applied to the case of edit distance, by a slight modification. However, the computation time will be much longer compared to Hamming distance, thus we believe that the Hamming distance is useful practically.

One of the disadvantage of our problem model is that we may have so many output pairs, and each pair does not give the shape of local similarity structures. Using the algorithm makes it possible to approach the problem of finding similar non-short strings, and similar non-short substrings of long strings. We can observe that two non-short similar strings may have several similar string pairs as their substrings. Thus, pairs of non-short strings including several similar string pairs are candidates for non-short similar

[1] http://research.nii.ac.jp/~uno/index.html

substrings. This approach has a certain certification of accuracy. For example, any two strings L_1 and L_2 of 3,000 letters with Hamming distance of at most 293 have at least three pairs of substrings of 30 letters starting from the same position of L_1 and L_2 such that the Hamming distance between them is at most two. We have similar observations for edit distance. For example, any two strings of 3,000 letters with edit distance at most 143 with insertions/deletions at most 55 includes at least three substrings of 30 letters with Hamming distance of at most two. This fact motivates us to find pairs of strings L_1 and L_2 of a middle length such that L_1 and L_2 have at least certain number of similar substrings such that the difference of their starting positions is bounded by some constant. We propose an algorithm for finding similar substrings composing such three pairs. In this way, we compared the human genome and mouse genome by our algorithm. The computation is done in quite short time and we could see homology structures figured out by the comparison.

The organization of this paper is as follows. The rest of this section shows some related works and applications to the problem. Section 2 is for preliminaries, and Section 3 shows our algorithm. Section 4 describes the filtering method for long string comparison. We show the computational experiments in Section 5, and conclude the paper in Section 6.

1.1 Related Works

In the area of algorithms and computation, the problem of finding similar strings has been widely studied. The problem is usually formulated that for two given strings Q and S, find all substrings of S similar to Q. This formulation can be considered as a generalization of string matching problems. When Hamming distance is chosen as a similarity measure, a straightforward algorithm solves the problem in $O(|S||Q|)$ time, thus a research goal is to reduce this time complexity. Here the length of S and Q is denoted by $|S|$ and $|Q|$.

For the problem of finding substrings of S with the shortest Hamming distance to Q, Abrahamson[1] proposed an algorithm running in $O(|S|(|Q|\log|Q|)^{1/2})$ time. If the maximum Hamming distance is k, the computation time can be reduced to $O(|S|(k\log k)^{1/2})$[4]. Some approximation approaches have been also developed. The Hamming distance of two strings of length l within $(1-\epsilon)$ and $(1+\epsilon)$ approximation ratio with probability δ can be computed in $O(\log l \log(1/\delta)/\epsilon)$ time [6]. For edit distance, which allows insertions and deletions, algorithms proposed by Muthukrishnan and Sahinalp[9, 10] approximate the minimum distance substring. Using these algorithms, the problem can be solved in shorter time but may fail with some solutions. These algorithms take more than $O(|S|^2)$ time to find similar substrings even for fixed length strings, Thus direct applications of these algorithms does not work in practice.

On the other hand, there are several studies for efficient data structures to find similar substrings. The problem is formulated such that, for a given string S, construct a data structure of not a large size such that for any query string Q, substrings of S similar to Q can be found in short time. For the problem of finding substring of S equal to Q, there are many efficient data structures such as suffix array which make it possible to find all such substrings in almost $O(|Q|)$ time. However, allowing the errors makes the problem difficult. Existing algorithms basically need $\theta(|S|)$ time in the worst case. This difficulty can be observed in many other similarity search problems, such as inner product of vectors, points in Euclidean space, texts and documents. Motivated by practical use, there have been many studies on approximation and heuristic approaches.

Yamada and Morishita [14] proposed an algorithm for computing a lower bound of the shortest Hamming distance from Q to a substring in S. The algorithm constructs a data structure in $O(|S|\log|S|)$ time, then answers a lower bound in $O(|Q|L)$ time for any Q, where L is a constant no greater than $|Q|$. They also proposed an efficient exact algorithm for strings with small alphabet such as genome sequences[15].

In bioinformatics area, the problem of finding substrings of two strings which are similar to each other is called homology search, and has been widely studied. In precise, for given two strings S and Q, the problem is to find pairs of a substring of S and a substring of Q which are similar to each other. Because of the huge size of genome sequences, developing exact algorithms running in short time is difficult thus many heuristic algorithms have been proposed. BLAST and FASTA[2, 3, 11] are widely used among these algorithms. The idea of BLAST is to find short substrings of S and Q that are equal and check whether there are similar substrings including them. This idea is based on the observation that

two similar substrings may have common short substrings. Actually, if the Hamming distance between two strings is no more than 9% of their length, they always have common string of 10 letters. The disadvantage of this method is that when the strings are long, huge number of substrings are the same, thus a lot of comparisons must be made. Such frequently appearing strings can be considered to be not important in practice, thus heuristic methods ignore these strings in the interest of practical efficiency. Another method of solving the problem is to partition Q and S into many blocks[13]. Some statistics of the blocks are computed, for example the number of each letter in the blocks, which for pruning blocks will never be similar. Then a dynamic programming connects the blocks and produces candidates of long similar substrings. The idea is that long similar substrings are expected to be not so many.

2 Preliminary

Let Σ be an alphabet of letters, and a *string* be a sequence of letters. The *length* of a string S is the number of letters in S and is denoted by $|S|$. A sequence composed of no letter is also a string and is called an *empty string*. The length of an empty string is 0. The ith letter of a string S is written $S[i]$, and i is called the *position* of $S[i]$. The substring of S starting from the ith letter and ending at the jth letter is denoted by $S[i,j]$. For example, when string S is $ABCDEFG$, $S[3] = C$, and $S[4,6] = DEF$. When $j < i$, we define $S[i,j]$ by the empty string. For two strings S_1 and S_2, the *concatenation* of S_2 to S_1 is a string S given by concatenating S_2 to S_1, i.e., $|S| = |S_1| + |S_2|$, $S[i] = S_1[i]$ if $i \leq |S_1|$, and $S_2[i - |S_1|]$ otherwise. The concatenation of S_2 to S_1 is denoted by $S_1 \cdot S_2$.

For two strings S_1 and S_2 of the same length, the *Hamming distance* of S_1 and S_2 is defined by the number of positions i satisfying that $S_1[i] \neq S_2[i]$. The Hamming distance is denoted by $HamDist(S_1, S_2)$. Such letters are called the *mismatch* of S_1 and S_2, and the positions of mismatches are called *mismatch positions* of S_1 and S_2. For a given threshold value d, we say two strings S_1 and S_2 of the same length are *similar* if their Hamming distance is no greater than d, and call them *similar string pair*. For string S and integers i and k, $i \leq k$, we denote the substring of S starting from $(\lceil |S|(i-1)/k \rceil + 1)$th letter to $(\lceil |S|i/k \rceil)$th letter, i.e., $S[\lceil |S|(i-1)/k \rceil + 1, \lceil |S|i/k \rceil]$, by $B(S,k,i)$. $B(S,k,i)$ is called the *ith block*.

For a string S, the *deletion* of the position i is a string given by $S[1,i-1] \cdot S[i+1,|S|]$. The *insertion* of letter a to S at position i is a string given by $S[1,i-1] \cdot A \cdot S[i,|S|]$ where A is the string composed of one letter a. The *change* of position i of S to a is a string given by $S[1,i-1] \cdot A \cdot S[i+1,|S|]$. For two strings S_1 and S_2, the *edit distance* of S_1 and S_2 is the smallest number of combinations of insertions, deletions and changes needed to transform S_1 to S_2.

The problem we address in this paper is formulated as follows. Let $\mathcal{S}$ be a multi set of strings of the same length. $\mathcal{S}$ is allowed to include more than one same string, and every string has an ID to be distinguished from the others. Hereafter we fix the input set $\mathcal{S}$ of strings of length l and a threshold value d, and assume that size of Σ is smaller than the size of $\mathcal{S}$.

3 Multi-sort Algorithm

The basic idea of the algorithm is to classify the strings in several ways so that any two similar strings are in the same group at least once. Let $C(k,j)$ be the set of j distinct integers taken from $1,\ldots,k$. For example, $C(4,2) = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\}$. For a string S and a set $C = \{i_1,\ldots,i_{k-d}\}$, $i_j < i_{j+1}$ taken from $C(k,k-d)$, we define $Sig(S,C) = B(S,k,i_1) \cdot B(S,k,i_2) \cdot \ldots \cdot B(S,k,i_{k-d})$. We suppose that an integer $k, d < k \leq l$ is chosen, and have a look at the following property.

Lemma 1. *If $HamDist(S_1,S_2) \leq d$, at least one $C \in C(k,k-d)$ satisfies $Sig(S_1,C) = Sig(S_2,C)$.*

Proof. The statement is obvious from the pigeonhole principle. Suppose that $HamDist(S_1,S_2) \leq d$. Observe that if $B(S_1,k,j) \neq B(S_2,k,j)$ holds, it includes at least one mismatch, i.e., $S_1[i] \neq S_2[i]$ holds for some i, $\lceil |S|(i-1)/k \rceil + 1 \leq i \leq \lceil |S|i/k \rceil$. Since S_1 and S_2 have at most d mismatches, at most d integers j satisfy $B(S_1,k,j) \neq B(S_2,k,j)$, thereby at least $k-d$ integers h satisfy $B(S_1,k,h) = B(S_2,k,h)$. Setting C to the set of those integers h satisfying $B(S_1,k,h) = B(S_2,k,h)$ shows that $Sig(S_1,C) = Sig(S_2,C)$. □

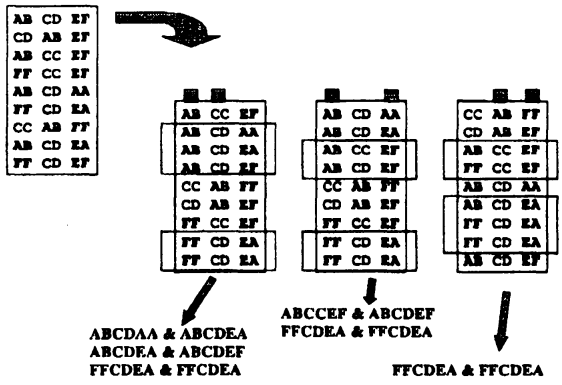

Fig. 1. Example of multi-sort for finding strings with Hamming distance of at most one, by dividing strings in three blocks and classifying them by two blocks.

This lemma motivates us to restrict the comparison to those pairs of strings satisfying the condition of the lemma. To efficiently find these pairs, we focus on the combinations of integers. For each $C \in C(k, k-d)$, we classify the strings S in $\mathcal{S}$ according to $Sig(S, C)$ so that two strings S_1 and S_2 satisfy $Sig(S_1, C) = Sig(S_2, C)$ if and only if they are in the same group. In Fig. 1, we show an example of this method, which we call the *Multi-sort Algorithm*. In the example, there are nine strings and set $d = 1$ and $k = 3$. Each block is composed of two letters, and classifications by two blocks are done three times. For each classification there are several groups represented by rectangles with more than one strings, and some of them contain strings with Hamming distance of at most one, written at the head of the arrows.

ALGORITHM MultiClassification_Basic ($\mathcal{S}$:set of strings of length l, d)
1. choose k from $d+1, \ldots, l$
2. **for each** $C \in C(k, k-d)$ **do**
3. classify all strings $S \in \mathcal{S}$ by $Sig(S, C)$
4. **for each** group K of the classification
output all pairs S_1 and S_2 in K satisfying $HamDist(S_1, S_2) \leq d$
6. **end for**

The classification for C is done by sorting $Sig(S, C)$ in $O(l(k-d)/k \times |\mathcal{S}|)$ time by a radix sort. We compute the probability that two randomly chosen letters from strings of $\mathcal{S}$ are the same, and choose k such that the expected size of each group in a classification is less than 1. Then the comparisons in a group is not so many, and the bulk of the computation time is for radix sort. Such a k can be chosen by computing the expected number of the size of each group. For example, if it is a small constant less than one, then the k can be considered to be sufficiently large. Since $l(k-d)/k$ is expected to be relatively small when l is small, it can be expected that the practical performance of the algorithm will be high.

4 Finding Non-short Similar Substrings

The algorithms proposed here is to detect similarity in short strings. In this section, we show an approach to detect non-short similar substrings based on short similar substring enumeration. A straightforward approach may take more than square time, since the length of strings to be compared can not be bounded by a constant.

One typical approach to capturing the similarity structures by using similar string pairs is as follows. We partition S into non-short blocks, for example, partition a string of 1,000,000 letters into 1,000 strings of 1,000 letters. We define the similarity measure of blocks $S[k_1, h_1]$ and $S[k_2, h_2]$ by the number of pairs of similar substrings taken from one block and a substring taken the other block. We can visualize the similarity structure in this measure by a figure such that the intensity of the color of the pixel (x, y) is

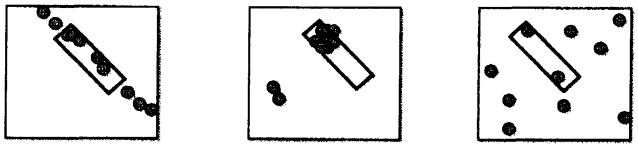

Fig. 2. Three example of cells having the same number of similar string pairs. Each dot represents the position of a string pair. The diagonal rectangle is the bounding condition to be a seed. When the threshold number is three, left and center cells have seeds. However, the seeds in the central cell is too much concentrated in a small area, thus we will remove the seeds.

given by the similarity. The left of Figure 4 shows an example of pictures obtained by this method. The figure is drawn by solving the problem with parameters $l = 30$ and $d = 2$. The computation is done in few minutes.

If the blocks are large, any two blocks have a sufficiently large number of similar string pairs, thus all pixels will be the same color. Moreover, we need much time for computation. In such cases, we have to reduce the number of output in some ways, without losing important information. One simple way to reduce the output is to output the pairs included in longer similar substrings. For example, we choose a constant k, and output a similar substring pair S_1 and S_2 only if S_1 and S_2 are substrings of L_1 and L_2 of length kl such that $S_1 = L_1[i, i+l-1]$ and $S_2 = L_2[i, i+l-1]$ hold for some i, and $HamDist(L_1, L_2) \leq kd$. We can also use the edit distance to be sensitive for insertion/deletion error.

Another way to reduce the output is sampling the substrings. For example, if we choose 1 of 10 substrings as substrings to be compared, we can reduce the number of output pairs possibly 1/100. However, This approach may miss some middle-length similar substrings. We here propose a way to sample the substrings which never miss similar substrings having a certain length.

Suppose that we are going to compare long string T_1 and T_2, by finding similar substrings for length l and Hamming distance at most d. We first choose a divisor of p. Then, we take substrings from T_1 such that their starting positions are $1, p+1, 2p+1, \ldots$, and take substrings from T_2 such that their starting positions are $1, 2, \ldots, p,\ l+1, l+2, \ldots, l+p,\ 2l+1, 2l+2, \ldots, 2l+p, \ldots$. An example of such a method of taking substrings is shown in the downside of Figure ??. We call this method *interleave method.* Suppose that L_1 and L_2 are substrings of T_1 and T_2 of length m such that $L_1 = T_1[i, i+m-1]$, $L_2 = T_2[j, j+m-1]$. Let $k = (i-j) \bmod l$, $x = p\lceil k/p \rceil$, and $y = k - (k \bmod p)$ Note that when $i-j < 0$, $k = l - ((j-i) \bmod l)$. Then, we can see that two substrings S_1 of L_1 and S_2 of L_2 of length l starting from $b+1$th letter, i.e., $S_1 = T_1[i, i+b+l-1]$ and $S_2 = T_1[j+b, j+b+l-1]$, are both taken to be compared if and only if $(i+b) \bmod l = x$, since $i+b \bmod l = x$ means that $j+b \bmod l = y$. Thus, for every l consecutive substrings pairs of length l, at least one pair satisfies that both substrings are taken for the comparison. Thus, intuitively, we never miss L_1 and L_2, if they are not sufficiently short and their Hamming distance is not large. In exact, we never miss L_1 and L_2 if their length is no less than $2l$, and the Hamming distance is less than $(d+1) \times \lfloor |L_1|/l \rfloor$. In the case that l has no divisor, or few divisors, we can choose a number $l' < l$ and its divisor p, and take strings from positions $1, 2, \ldots, p,\ l'+1, l'+2, \ldots, l'+p,\ 2l'+1, 2l'+2, \ldots, 2l'+p, \ldots$. In this way, we lose the above certification, but expect that the practical efficiency does not change so much.

The second approach is based on an observation that non-short similar substrings have several similar short substrings. Observe that any two strings L_1 and L_2 sufficiently longer then l with Hamming distance less than $|L_1| \times d/l$ must include several similar string pairs. For example, when $l = 30, d = 2$, and L_1 and L_2 are of 3,000 letters with Hamming distance of at most 293, they include at least three similar string pairs. This comes from the same reason as Lemma 1. We call such pairs a *seed.* It implies that there are long similar strings with over 3000 letters only if there is a seed. This motivates us to find seeds to capture the long string similarity; draw an image by putting a dot if there are such three pairs. Further, if the member of a seed lies in a short interval, say 300, and have no other similar string pair exists closed to the members, then the seed indicates short similar strings, thus we can also such isolated seeds lies in a small area. To find such pairs, we classify all similar string pairs according to the difference of the starting position of two strings in the pair. Then, we sort the similar pairs having the same difference of the starting positions according to the starting position of the first string. Then, by scanning the obtained

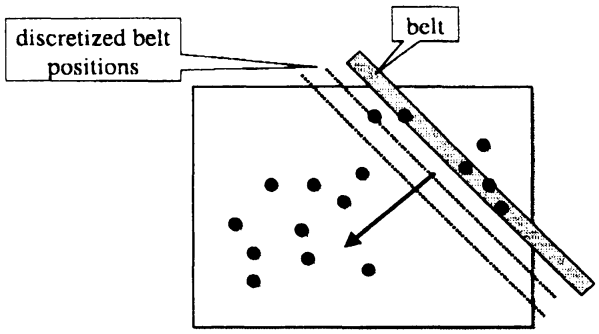

Fig. 3. An example of a belt; it sweeps in left-down direction. It updates the sorted order of containing string pairs, and find the pairs satisfies the condition to be a seed. Discretized belt, having doubled width, is placed only at the dotted lines.

sorted list of pairs, we can easily find seeds. Actually, the classification and the sort of each group can be done by a radix sort in linear time, this task can be done in $O(N)$ time where N is the number of similar pairs.

This approach can be applied even when we consider edit distance. An insertion/deletion can make the Hamming distance of two strings quite large. Thus, if the number of insertions/deletions between two strings is relatively small, we can state a certain certification of accuracy. Consider an example of two strings of 3,000 letters with edit distance at most 198 with insertions/deletions at most 55. Then, they have at least three substrings of 30 letters with Hamming distance of at most two. In the case of edit distance, the difference of the start positions of the pairs in a seed do not have to be the same, but the difference is bounded, at most 55 in this case. In Figure 2, we present some examples of similar string pairs in cells. Some bounding conditions of seeds are drawn by diagonal boxes.

To find all such three pairs in this case, we modify the above method. The starting positions of pairs in a seed can differ, thus we have to merge several groups having similar starting position difference, then sort the pairs and scan it. We call the merged groups *belt*. An intuitive image of the belt is drawn in Figure 3. Thus, the computation time is multiplied by the width of the belt, by we can reduce the computation time by using binary tree. We construct a binary tree representing the sorted order of the pairs in a belt, then we shift the belt by removing one group and adding on group to the belt, and by using binary tree, we update the sorted list of the pairs in the belt. In this way, the computation time to find all pairs not included in such three pairs can be done in $O(N \log N)$ time.

In practice, we can use more simple method by discretizing the belts. We double the width w of the belt, and scan only one belt among w belts. An example of the positions of belt is shown in Figure 3. It makes the computation time to linear, but it never miss any seed, but some similar string pairs included in no seed may be judged to be in a seed. However, we can expect such error is not critical, and does not matter, since we can consider that random noise can make few such error. If there are many such errors, we should consider such error as a kind of similarity. We display a figure made by this approach in the right of Figure 4. The figure is drawn by first finding the problem with parameter $l = 30$ and $d = 3$, and find seeds composed of three pairs with length 3000 and width 300, by the discrete belt approach. We discard the isolated seeds, if a seed is in an area of length 300, and has no similar string pair with distance shorter than 2700, in the belt. The resolution of the figure is 2000 by 2000, and each dot is written when it has at least two seeds. Each dot is enlarged to be emphasized.. We successfully removed the noise patterns from the picture and emphasized the similar structures. The computation is also done in few minutes.

5 Computational Experiments

This section shows the results of computational experiments of our algorithm. The code was written in C, and compiled with gcc. We used a note PC with a Pentium M 1.2GHz processor with 768 MB of

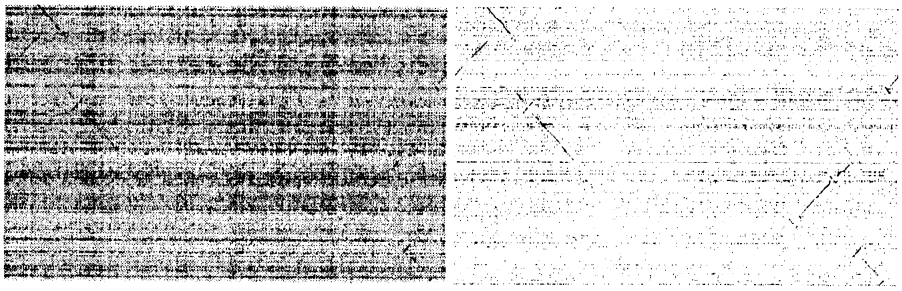

Fig. 4. Matrix showing similarity of mouse 11 chromosomes (X-axis) and Human 17 chromosome (Y-axis), with black cells on similar parts; we can see similar substructures as diagonal lines; left figure represents the density of similar string pairs, and the right figure is the result of our filtering.

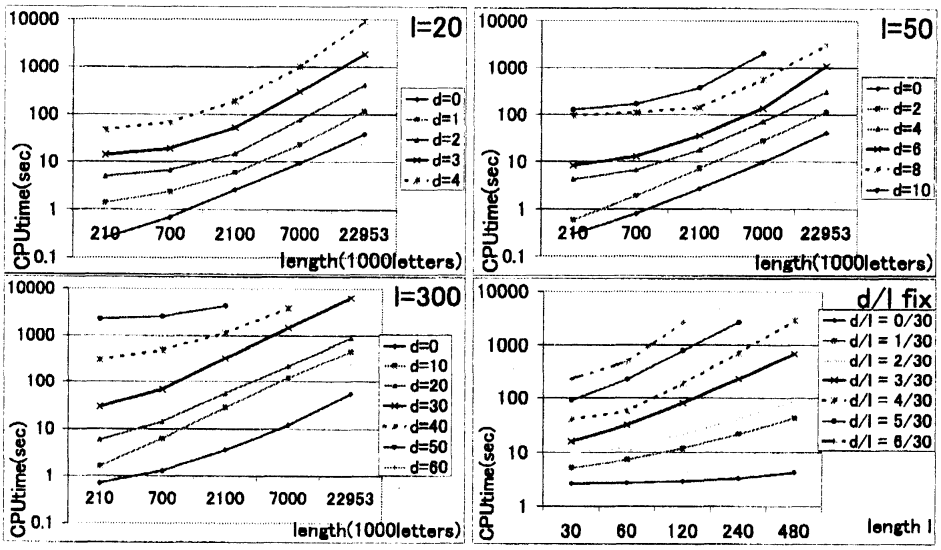

Fig. 5. Increase in computation time against the increase in database size with fixed l and d: the right-lower figure is for fixed d/l inputting a string of 2.1 million letters

memory, with cygwin which is a Linux emulator on Windows. The implementation is available at the author's homepage; http://research.nii.ac.jp/~uno/index.html.

The instance is the set of substrings of fixed length taken from the Y chromosome of Homo sapiens. The length is set to 20, 50 and 300. Figure 5 shows the results. Each line corresponds to one threshold value d. The X-axis is the number of input substrings, and Y-axis is the computation time. Both axes use log scales. We can see that the computation time increases slightly higher than linear, but smaller than the square.

We also show the increase in computation time against the increase of l with fixed d/l. The instance is fixed to that with 2.1 million strings, and the result is shown in the right-lower figure of Figure 5. The left of Figure 6 shows the number of executed radix sorts, which means the number of recursive calls. The horizontal axis is for the length of the input string, and the vertical axis is for the number of recursive calls. Each line corresponds to the result of fixed length l and threshold d. In this implementation, when the members in a group is sufficiently small, we execute the pairwise comparison immediately and do not execute further recursive calls. The result shows the number of recursive calls is also robust for the increase of the length, when the length and the threshold is fixed. From these results, at least for genome sequences our algorithm is quite scalable for the increase of input string.

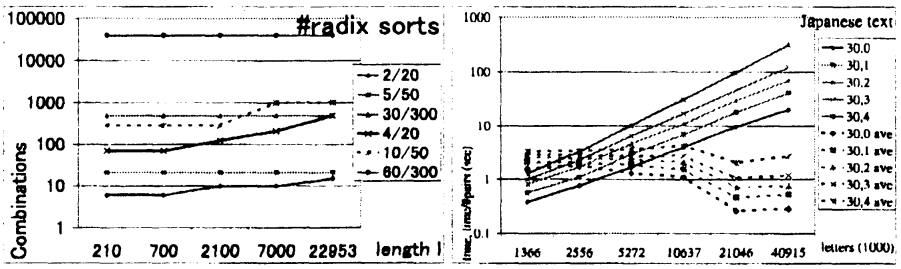

Fig. 6. (left) number of radix sorts performed, (right) experiments for Japanese web texts

The right of Figure 6 is the experimental result for Japanese texts taken from Web pages, crawled at 2007. The data is collected by Kawahara and Kurohashi[7]. The horizontal axis is the length of input data in log scale, and the vertical axis is the CPU time, and CPU time per one million output pairs, in second in log scale. The lengths of input are from 1.3 million to 40 million. On average each page has 1,200 letters, and the size of alphabet is about 8,000. We fixed the length l to 30, and evaluate the increase of the computation time for each $d = 0, \ldots, 4$. This experiments are done on a PC with Intel Core 2 Duo E8400 (3.0GHz) with 4GB memory, with Linux and gcc. The computation time per pair found does not increase against the increase of the input length, thus we would say our multi-sort algorithm also scales for this data.

6 Conclusion

We proposed an efficient algorithm for enumerating all pairs of strings with Hamming distance at most given d from string set $\mathcal{S}$. We proposed multi-sort algorithm whose computation time is practically linear time. We proved that the computation time of its variant is bounded by linear of the string length when the length of strings in the string set is constant. A simple modification of the algorithm adapts the edit distance, and computation of mismatch tolerance. A new similarity continuous interval Hamming distance is introduced to define maximal similar substrings clearly.

We proposed a method of finding similar non-short substrings from huge strings. We modeled similar non-short strings by two non-short strings including several short similar substrings. We presented an efficient algorithm for finding those strings from huge strings. By the computational experiments for genome sequences, we demonstrated the practical efficiency of the algorithm, and the efficiency of the parallelization. On the comparison of genome sequences, we could find similar long substrings from human and mouse genomes in a practically short time.

Acknowledgments

We gratefully thank to Professor Asao Fujiyama of National Institute of Informatics of Japan, Professor Shinichi Morishita of Tokyo University Doctor Takehiko Itoh of Mitsubishi Research Institute, and Professor Hidemi Watanabe of Hokkaido University, for their valuable comments. We would also like to thank to Professor Tsuyoshi Koide and Doctor Juzo Umemori of National Institute of Genetics for their contribution to the evaluation of the algorithm on practical genome problems. For the parallelization of the implementation, we would like to thank Yasuhiro Ike of Ybeat, Japan, for his help.

References

1. K. Abrahamson, Generalized String Matching, *SIAM Journal on Computing*, **16(6)**, pp. 1039–1051, 1987.
2. S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, Basic local alignment search tool, *Journal on Molecular Biology* **215**, pp. 403–10, 1990.

3. S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang Z, W. Miller, D. J. Lipman, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs, *Nucleic Acids Research*, **25**, pp. 3389–3402, 1997.
4. A. Amir, M. Lewenstein, and E. Porat, Faster Algorithms for String Matching with k Mismatches, *In Symposium on Discrete Algorithms*, pp. 794–803, 2000.
5. P. Brown and D. Botstein, Exploring the New World of the Genome with DNA Microarrays, *Nature Genetics*, **21**, pp. 33–37, 2000.
6. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan, An Approximate l1-difference Algorithm for Massive Data Streams, *In Proceedings of FOCS99*, 1999.
7. D. Kawahara and S. Kurohashi, Case Frame Compilation from the Web using High-Performance Computing, *In Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC2006)*, pp. 1344–1347, 2006.
8. U. Manber and G. Myers, Suffix Arrays: A New Method for On-line String Searches, *SIAM J. on Comp.*, **22**, pp. 935–948, 1993.
9. S. Muthukrishnan and S. C. Sahinalp, Approximate Nearest Neighbors and Sequence Comparison with Block Operations, *In Proceedings of 32nd annual ACM symposium on Theory of Computing*, pp. 416–424, 2000.
10. S. Muthukrishnan and S. C. Sahinalp, Simple and Practical Sequence Nearest Neighbors under Block Edit Operations, *In Proceedings of CPM2002*, 2002.
11. W. R. Pearson, Flexible sequence similarity searching with the FASTA3 program package, *Methods in Molecular Biology* **132**, pp. 185–219, 2000.
12. K. Popendorf, Y. Osana, T. Hachiya, and Y. Sakakibara, Murasaki - homology detection across multiple large-scale genomes, Fifth Annual RECOMB Satellite Workshop on Comparative Genomics, San Diego, USA, 2007.
13. S. Yamada, O. Gotoh, H. Yamana, Improvement in Accuracy of Multiple Sequence Alignment Using Novel Group-to-group Sequence Alignment Algorithm with Piecewise Linear Gap Cost, *BMC Bioinformatics* **7**, pp. 524, 2006.
14. T. Yamada and S. Morishita, Computing Highly Specific and Mismatch Tolerant Oligomers Efficiently, *Bioinformatics Conference 2003*, 2003.
15. T. Yamada and S. Morishita, Accelerated Off-target Search Algorithm for siRNA, *Bioinformatics* **21**, pp. 1316–1324, 2005.